High-Performance Computing for Aerial Autonomy: GPU-Driven Training of Multirotor Vehicles

Anthony Skenter

***Abstract*—This research presents FlyBy, a novel high-performance reinforcement learning framework specifically designed for training autonomous quadrotor vehicles. Traditional RL frameworks prioritize ease of use over computational efficiency, creating significant bottlenecks for complex applications such as Co-Design where agents must be trained rapidly across diverse configurations. FlyBy addresses these limitations through two primary innovations: enhanced learning techniques and GPU acceleration strategies.**

The framework integrates Blade Element Momentum (BEM) theory for high-fidelity aerodynamic modeling, providing more accurate thrust and torque predictions compared to simplified quadratic models. A sophisticated curriculum learning strategy progressively increases task complexity across eight difficulty levels, enabling agents to master complex figure-8 trajectories through systematic skill development. The reward function engineering incorporates multiple components including progress tracking, velocity alignment, and completion bonuses, effectively balancing immediate feedback with long-term objectives.

GPU acceleration is achieved through comprehensive JAX implementation, leveraging just-in-time compilation, automatic vectorization, and parallelization.

The environment simulation processes up to 160 parallel instances simultaneously, achieving over 150,000 steps per second on consumer hardware—a dramatic improvement over traditional CPU-based approaches. The neural network architecture employs an actor-critic design with shared feature extraction and diagonal Gaussian policies optimized for continuous control tasks.

Experimental results demonstrate significant performance improvements across multiple metrics. GPU acceleration provides up to 11× speedup compared to baseline implementations, while curriculum learning enables successful policy convergence where direct training at maximum difficulty fails entirely. The BEM lookup table implementation achieves 150-300× computational speedup over full aerodynamic calculations while maintaining prediction accuracy above 99%. Learning rate annealing strategies show superior convergence properties to higher episode rewards.

The framework's effectiveness is validated through comprehensive ablation studies examining entropy coefficients, episode length constraints, and target distance variations. Results indicate optimal entropy values around 0.01, with longer episode limits enabling higher performance through improved task completion. The research demonstrates that appropriate task difficulty calibration is critical, with exponentially increasing complexity beyond certain thresholds.

FlyBy represents a significant advancement in computational efficiency for aerial vehicle training, reducing training times from hours to minutes while maintaining or improving policy quality. The framework's modular design and GPU optimization techniques provide a foundation for future research in high-performance reinforcement learning applications, particularly relevant for co-design optimization and real-time autonomous systems development.

This work was carried out as a final-year project in the Department of Aeronautics, Imperial College London (2024/2025). The author is no longer affiliated with the institution.

***Index Terms*—Reinforcement Learning, GPU Acceleration, Quadrotor Control, Blade Element Momentum Theory, Curriculum Learning, JAX, High-Performance Computing**

I. Introduction

A. Motivation

As Frank Lloyd Wright said in 1908, "Form follows function-that has been misunderstood. Form and function should be one, joined in a spiritual union". If one considers form and function independently then the outcome will always be monolithic, far from optimal. Nature always considers the union of these two entities, hence maximizing efficiency. Natural selection is the mechanism serving Nature's purpose of evolving form and function.

Co-Design of robotic agents is the principle which enables the simultaneous design of form, function mimicking Nature's recipe [1]. Each agent is trained to accomplish specific tasks inside a domain. Then the characteristics of the best performing agents are mutated to next generations, hence creating "better" offsprings. Since form depends on function and vice versa, there is not a systematic way to a priori model the favourable characteristics of an agent. In a sense Co-Design explores a brute force approach to capture the best traits of each agent and evolve them.

In practice Co-Design of agents is a multi-objective optimization problem [2]. Agents are initially trained using an actor-critic neural network, which acts as the "brain". This training is performed against high level user defined objectives. Then, algorithms such as Non-dominated sorting genetic algorithm II (NSGA-II) [3] are usually employed to evolve the form of the agents with random mutations. Recent applications of Co-Design optimization to morphing wing drones have demonstrated significant improvements in energy efficiency and mission completion time compared to fixed-wing counterparts [4].

Learning is the ability of agents to acquire new knowledge and skills from their experience. Animals learn by receiving feedback from their environment upon their actions with operant conditioning [5]. Actions or inputs yielding favourable outcomes or outputs, receive positive feedback or rewards. In other words, actions yielding favorable outcomes are systematically reinforced, increasing their selection probability. Reinforcement Learning (RL) represents one of the three fundamental paradigms of machine learning alongside supervised and unsupervised learning. It provides a framework through which machines learn optimal behaviors by taking actions that maximize cumulative rewards in an environment, mirroring

how humans and animals acquire skills through experience and feedback.

Over the past years there have been remarkable breakthroughs in RL, particularly in robotics. Deepmind AlphaGo and AlphaZero, demonstrated that agents can optimize against high order objectives and achieve better performance than humans [6].

While RL optimizes behavioral policies within fixed structures, Co-Design introduces morphological adaptation through evolutionary algorithms. The process of training the agents and then mutating their characteristics is not sample efficient and requires heavy computational resources. Agent training duration depends on the dimensionality of the state space, complexity of objective functions and the domain of operation. In non-linear dynamic environments, the intricate cause-and-effect relationships exhibit path dependence and emergent behaviors that extend training time requirements for task mastery. Hence, agents need to explore a much wider range of scenarios and learn to generalize across highly variable conditions.

B. Aims

The purpose of this research is to create a modern RL environment which adheres to high performance computing principles and is faster than any other current open-source library. RL frameworks such as Stable Baselines 3(SBL3), Open AI Gym, MuJoCo are designed for ease of use rather than raw performance. These libraries don't work well in Co-Design problems as they are significantly slow.

FlyBy, a novel RL framework is proposed for training a Quadcopter to accomplish different tasks. In particular, the quadcopter accomplish missions such as path tracking given some high order objectives. A novel addition to this environment is Blade Element Momentum (BEM) theory. Using BEM the actual forces produced by the blade element geometry are calculated yielding a better representation of the flight dynamics. Critical components of RL training procedures are examined, with particular focus on the exploration vs exploitation dilemma. Different modifications to exploration strategies, reward function design, and sample efficiency are analysed to enhance training outcomes. By implementing techniques such as intrinsic motivation, curriculum learning, and prioritized experience replay, the research demonstrates measurable improvements in both convergence speed and final policy performance across diverse environments.

FlyBy's computational capabilities are expanded through GPU acceleration, leveraging parallel processing architectures, aiming to dramatically reduce agent training time. By offloading computationally intensive neural network operations to specialized graphics hardware, this implementation enables simultaneous execution of thousands of calculations, addressing the computational bottleneck in non-linear environment simulation. This optimization significantly enhances FlyBy's training efficiency without compromising performance quality.

C. Report Overview

This report is structured to systematically address the research aims. Section 2 establishes the theoretical foundation, covering quadrotor dynamics, classical control approaches, and reinforcement learning fundamentals. Section 3 introduces the FlyBy framework, detailing its architecture, implementation specifics, and enhanced learning techniques. It also focuses on GPU acceleration strategies employed to optimize training performance. Section 4 presents performance results and analysis. Finally, Section 5 concludes with a summary of contributions and directions for future work.

II. Theoretical Foundation

A. Quadrotor Dynamics

1) System Dynamics: The quadrotor model follows rigid body dynamics with six degrees of freedom (6-DOF). The state of the quadrotor is represented by a 12-dimensional vector including position, linear velocity, orientation (Euler angles), and angular velocity:

$$\mathbf{s} = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, p, q, r]^T \tag{1}$$

where (x, y, z) represents the position in the inertial frame, $(\dot{x}, \dot{y}, \dot{z})$ are linear velocities, (ϕ, θ, ψ) are roll, pitch, and yaw angles, and (p, q, r) are angular velocities.

The dynamics of the quadrotor are governed by the following equations:

$$\begin{array}{cc} \dot{\mathbf{p}} = \mathbf{v} & \dot{\mathbf{v}} = \mathbf{g} + \frac{1}{m}\mathbf{R}(\mathbf{\Phi})\mathbf{T} \\ \dot{\mathbf{\Phi}} = \mathbf{W}(\mathbf{\Phi})\omega & \dot{\omega} = \mathbf{I}^{-1}(\tau - \omega \times \mathbf{I}\omega) \end{array} \tag{2}$$

where $\mathbf{p} = [x, y, z]^T$ is the position, $\mathbf{v} = [\dot{x}, \dot{y}, \dot{z}]^T$ is the linear velocity, $\mathbf{g} = [0, 0, -g]^T$ is the gravity vector, m is the mass, $\mathbf{R}(\mathbf{\Phi})$ is the rotation matrix from body to inertial frame, $\mathbf{T} = [0, 0, T]^T$ is the thrust vector in the body frame, $\mathbf{\Phi} = [\phi, \theta, \psi]^T$ are the Euler angles, $\mathbf{W}(\mathbf{\Phi})$ is the transformation matrix from angular velocities to Euler angle rates, $\omega = [p, q, r]^T$ are the angular velocities, $\mathbf{I}$ is the inertia matrix, and τ is the torque vector.

The rotation matrix $\mathbf{R}(\mathbf{\Phi})$ converts vectors from the body frame to the inertial frame and is defined as:

$$\mathbf{R}(\mathbf{\Phi}) = \begin{bmatrix} c_\psi c_\theta & c_\psi s_\theta s_\phi - s_\psi c_\phi & c_\psi s_\theta c_\phi + s_\psi s_\phi \\ s_\psi c_\theta & s_\psi s_\theta s_\phi + c_\psi c_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{3}$$

where $c_\alpha = \cos(\alpha)$ and $s_\alpha = \sin(\alpha)$ for any angle α.

The transformation matrix $\mathbf{W}(\mathbf{\Phi})$ maps the body angular rates to Euler angle rates:

$$\mathbf{W}(\mathbf{\Phi}) = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix} \tag{4}$$

The total thrust T and torques $\tau = [\tau_x, \tau_y, \tau_z]^T$ are related to the individual motor forces f_i through:

$$\begin{array}{cc} T = \sum_{i=1}^{4} f_i & \tau_x = l(f_1 - f_3) \\ \tau_y = l(f_2 - f_4) & \tau_z = c(f_1 - f_2 + f_3 - f_4) \end{array} \tag{5}$$

where l is the arm length and c is the drag-to-thrust ratio.

The motor dynamics are modeled as a first-order system:

$$\dot{\omega}_m = \frac{1}{\tau_m}(\omega_{m,des} - \omega_m) \tag{6}$$

where ω_m is the motor speed, $\omega_{m,des}$ is the desired motor speed, and τ_m is the motor time constant (typically 0.05-0.15 seconds).

The relationship between motor speed ω_m and thrust f is approximated as:

$$f_i = k_f \omega_{m,i}^2 \tag{7}$$

where k_f is the thrust coefficient.

In the implementation, a semi-implicit Euler integration method with a time step of $\Delta t = 0.02$ seconds is used, this provides a good balance between simulation accuracy and computational efficiency.

2) Blade Element Momentum Aerodynamics: Instead of the common quadratic approximation $T = k_T \omega^2$, the environment models per-rotor thrust and torque with Blade Element Momentum (BEM) theory, which couples momentum theory with blade-element analysis to capture blade geometry, airfoil behaviour and inflow conditions, building on classical rotor aerodynamics [7] and recent high-fidelity quadrotor BEM models [8]. The blade is discretised into annular elements and the axial and tangential induction factors are obtained by fixed-point iteration with Prandtl tip- and hub-loss corrections; the high-induction regime ($a > 0.4$) uses the Buhl/Burton empirical correction [9]. Because this iteration is too expensive to run inside the training loop, the converged thrust and torque are precomputed over a grid of axial velocity and rotor speed and queried at runtime by bilinear interpolation. The resulting model is validated against experimental propeller data in the results.

B. Reinforcement Learning Background

We cast quadrotor control as a Markov decision process $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, in which a stochastic policy $\pi_\theta(a \mid s)$ is optimised to maximise the expected discounted return $\mathbb{E}[\sum_t \gamma^t r_t]$. FlyBy uses Proximal Policy Optimization (PPO), an on-policy actor–critic algorithm that maximises a clipped surrogate objective to keep each update close to the data-collection policy, with Generalized Advantage Estimation (GAE) for the advantage targets. PPO is widely used for continuous control because of its stability and sample efficiency.

Deep RL has been applied to quadrotor stabilisation, waypoint navigation and agile racing [10], [11], [12], typically with either low-level (motor/body-rate) or mid-level (collective-thrust-and-attitude) action spaces; FlyBy acts at the collective-thrust-and-body-torque level. A recurring obstacle is the high computational cost of training, which motivates the GPU-accelerated design described in this work.

C. Deep Reinforcement Learning Algorithms for Quadrotor Control

Several state-of-the-art DRL algorithms have been successfully applied to quadrotor control tasks:

1) Proximal Policy Optimization (PPO): PPO is an on-policy algorithm that has been widely used for quadrotor control due to its stability and simplicity. The PPO objective function is:

$$\begin{aligned} L^{PPO}(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \\ - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\big] \end{aligned} \tag{8}$$

where L^{VF} is a value function loss and S is an entropy bonus that encourages exploration.

The clipping operation in PPO effectively constrains the policy update to remain within a trust region defined by the parameter ϵ, typically set to 0.1 or 0.2. This prevents large policy changes that could destabilize training while maintaining computational efficiency compared to the exact trust region constraint used in TRPO. Schulman et al. [13] demonstrated that this clipping approach provides similar performance to TRPO while being significantly simpler to implement and more compatible with architectures that include noise or parameter sharing (critical for GPUs).

III. Methodology

A. FlyBy: A High-Performance RL Framework

1) Framework Design:

a) Architecture Overview: Figure 1 gives an overview of the complete training and inference pipeline. FlyBy is designed as a comprehensive reinforcement learning framework specialized for aerial vehicle training. Its architecture consists of four core components:

- **Simulation Environment**: A high-fidelity quadrotor physics model
- **Policy Network**: A deep neural network for state-to-action mapping
- **Training Engine**: An optimized implementation of PPO with experience collection
- **Curriculum Manager**: A system for adaptive task difficulty progression

These components are integrated into a modular, extensible framework that allows for straightforward experimentation with different architectural choices, hyperparameters, and training strategies.

2) Environment Implementation: The core of FlyBy is a specialized quadrotor environment that faithfully models the dynamics of a quadrotor drone. Key features include:

- **State Space**: 16-dimensional observation vector including position, linear velocity, orientation (Euler angles), angular velocity, direction to target waypoint, and current waypoint index
- **Action Space**: 4-dimensional continuous action space representing collective thrust and torques around the three rotational axes
- **Physics Model**: 6-DOF rigid body dynamics with semi-implicit Euler integration
- **Task Design**: Waypoint-based navigation with configurable trajectories and thresholds

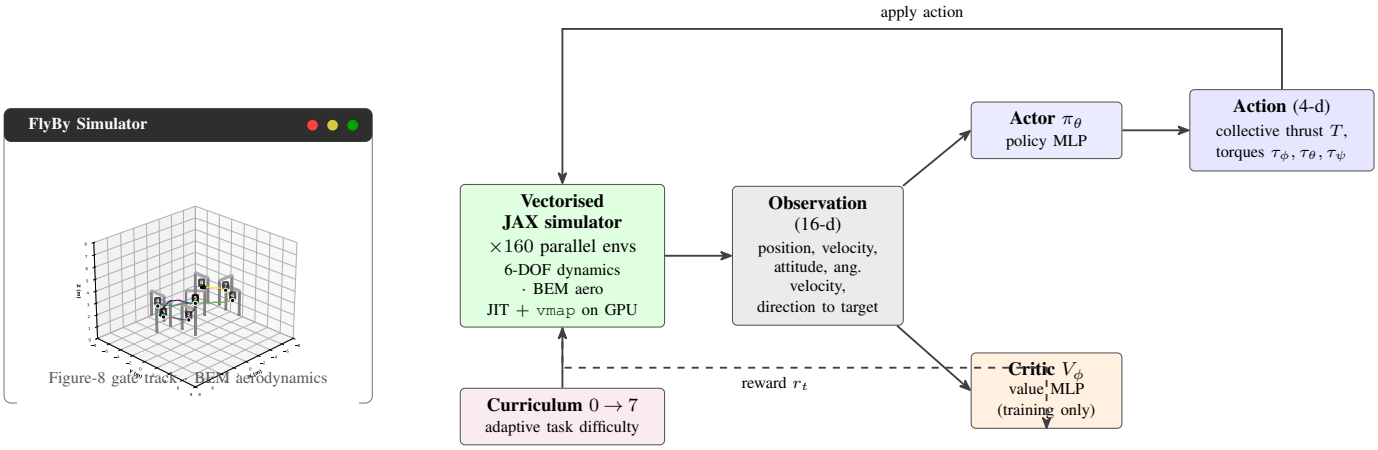


Fig. 1: Overview of the FlyBy training and inference setup with a view of the simulator UI (left). At each step the vectorised JAX simulator—running 160 parallel environments with Blade Element Momentum aerodynamics, just-in-time compiled on the GPU—emits a 16-dimensional observation. The actor π_θ maps it to a 4-dimensional action (collective thrust and the three body torques), which is applied back to the simulator. During training the critic V_ϕ additionally estimates the value function from the reward signal and a curriculum manager progressively raises the task difficulty ($0 \to 7$); at inference only the actor is deployed.

The environment is implemented as a subclass of the Gymnax environment interface, allowing seamless integration with standard RL libraries while maintaining compatibility with JAX for GPU acceleration.

The dynamics implementation in FlyBy directly applies thrust and torques to the rigid body model without translating them to individual motor commands through a control allocation mixer. This design decision was deliberate, as it eliminates an unnecessary layer of complexity in the learning process. The _apply_dynamics method takes the normalized thrust and torque values from the agent's action, scales them to physical units, and directly applies them in the physics simulation:

```
def _apply_dynamics(self, thrust, torques):
    # Rotation matrix calculation
    R = self._rotation_matrix(angles)

    # Forces (in inertial frame)
    thrust_force = R @ np.array(
        [0, 0, thrust * self.max_thrust])
    gravity_force = np.array(
        [0, 0, -self.mass * self.g])
    total_force = thrust_force + gravity_force

    # Accelerations
    linear_acc = total_force / self.mass
    angular_acc = torques * self.max_torque / self.I

    # State update with semi-implicit Euler
    integration
    # ...
```

This approach focuses the learning problem on the high-level control aspects (thrust and torque generation) rather than the lower-level motor control details. This aligns with the common practice in RL of providing the right level of abstraction to make the learning problem tractable while preserving the essential dynamics of the system.

The simulation time step is set to 0.02 seconds (50 Hz), balancing simulation accuracy with computational efficiency. This rate is high enough to capture the essential dynamics of quadrotor motion while maintaining reasonable training speeds.

3) Aerodynamic Model: The environment couples the rigid-body dynamics to the BEM aerodynamic model introduced above. At each step the per-rotor axial inflow is computed from the body velocity and rotor kinematics, and the corresponding thrust and torque are read from the precomputed BEM table by bilinear interpolation; the resulting forces and moments are summed and integrated with semi-implicit Euler. This retains the essential aerodynamic effects (such as thrust variation in forward flight) while remaining cheap enough for large-scale parallel training.

4) RL Algorithm Implementation: While traditional control approaches described in Section 2.2 provide a solid foundation for quadrotor control with well-understood stability guarantees, they face limitations in handling complex dynamics, high-dimensional state spaces, and uncertainty. Classical PID and cascaded control architectures require precise system identification, manual tuning, and often struggle with strongly coupled nonlinear dynamics. FlyBy addresses these limitations by implementing an RL approach that learns optimal control policies directly from experience without requiring explicit mathematical models. Unlike cascaded control architectures that separate position and attitude control loops, RL enables end-to-end learning of control policies that can optimize directly for high-level objectives while adapting to the system's dynamics. This transition from model-based to learning-based control represents a fundamental shift from explicit controller design to implicit policy learning. Though both approaches can be complementary—classical controllers can provide initial policies or safety guarantees while RL optimizes for performance in regions where analytical solutions fall short.

FlyBy implements a JAX-based PPO algorithm for training the quadrotor agent. PPO was selected for its stability, sample efficiency, and strong performance on continuous control tasks.

Algorithm 1 PPO Objective Calculation

Collect trajectories (s_t, a_t, r_t, s_{t+1}) by running current policy $\pi_{\theta_{old}}$
Calculate value estimates $V(s_t)$ for all states in collected trajectories
for each timestep t **do**
    Calculate TD error: $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
    Compute GAE advantage: $\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$
end for
for each optimization epoch **do**
    for each minibatch of experience **do**
        Compute probability ratio: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
        Calculate clipped objective: $L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$
        Compute value function loss: $L^{VF}(\theta) = \mathbb{E}_t[(V_\theta(s_t) - V_t^{target})^2]$
        Calculate entropy bonus: $S[\pi_\theta] = \mathbb{E}_t[-\log \pi_\theta(a_t|s_t)]$
        Compute total loss: $L^{Total}(\theta) = L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta]$
        Update network parameters using gradient descent on $L^{Total}(\theta)$
    end for
end for

5) Neural Network Architecture: An actor-critic network architecture was implemented using Flax, a neural network library designed for JAX:

$$\pi_\theta, V_\theta = \text{ActorCritic}(s) \tag{9}$$

$$h_i = \tanh(W_i h_{i-1} + b_i) \quad \text{for } i \in \{1, 2, 3, 4\} \tag{10}$$

$$\mu_\theta = W_\mu h_4 + b_\mu \tag{11}$$

$$\log \sigma_\theta = \theta_{\log \sigma} \tag{12}$$

$$V_\theta = W_V h_4 + b_V \tag{13}$$

where $h_0 = s$ is the input state, h_i are hidden layers with 160 units each, μ_θ is the mean of the action distribution, $\sigma_\theta = \exp(\theta_{\log \sigma})$ is the standard deviation, and V_θ is the state value estimate.

Key features of network design include:

- Four hidden layers with 160 units each for both actor and critic
- Tanh activation functions for stable gradient propagation
- Orthogonal initialization with carefully tuned scaling factors
- Shared feature extraction layers between policy and value functions
- Diagonal Gaussian policy with state-independent, learnable log standard deviations

6) PPO Algorithm: The PPO implementation follows the clipped objective variant [14]:

with the following hyperparameters:

- Clipping parameter $\epsilon = 0.2$
- Discount factor $\gamma = 0.99$
- GAE parameter $\lambda = 0.95$
- Learning rate = 8×10^{-4} with linear annealing
- 4 epochs per update with 4 minibatches per epoch
- Value function coefficient $c_1 = 0.5$ and entropy coefficient $c_2 = 0.01$

7) Training Process: The training process consists of alternating data collection and policy update phases:

Algorithm 2 FlyBy PPO Training Loop

Initialize actor-critic network θ
Initialize environment at lowest difficulty level
for $n = 1$ to $NumUpdates$ **do**
    Collect trajectories by running policy in environment for $NumSteps$ steps
    Compute advantages using Generalized Advantage Estimation (GAE)
    for $e = 1$ to $NumEpochs$ **do**
        Shuffle collected data
        for each minibatch **do**
            Compute clipped PPO objective
            Compute value function loss
            Compute entropy bonus
            Update network parameters using Adam optimizer
        end for
    end for
    Calculate success rate over recent episodes
    if success rate $>$ threshold and not at maximum difficulty **then**
        Increase environment difficulty level
        Reset environment with new difficulty
    end if
end for

This training procedure incorporates curriculum learning by automatically adjusting the difficulty based on the agent's performance, allowing it to progressively master increasingly complex tasks.

8) Enhanced Learning Techniques: The development of FlyBy's enhanced learning techniques draws inspiration from classical control theory while addressing its limitations through modern RL advancements. Where classical control relies on manually designed feedback loops with specific gains (as described in Section 2.2.3). The enhanced RL approach automatically discovers optimal feedback mecha-

TABLE I: Curriculum Difficulty Levels

Level	Description	Waypoints
0	Single waypoint at fixed height, generous threshold (0.8m)	1
1	Three waypoints forming a partial circle	3
1.3	Four waypoints with additional height stabilization rewards	4
1.5	Four waypoints forming a wider partial figure-8 pattern	4
1.7	Five waypoints with slightly relaxed positioning requirements	5
2	Full half figure-8 pattern with five waypoints	5
3	Complete figure-8 trajectory	8
4	Complex figure-8 with precisely positioned waypoints	12

nisms through experience. The reward function engineering described below incorporates domain knowledge similar to how a control engineer would design cost functions for optimal control, but allows for more complex, non-quadratic objectives that better capture the full spectrum of desirable behaviors. Additionally, the curriculum learning strategy parallels gain scheduling techniques from classical control, where controller parameters are adapted based on operating conditions, but extends this concept to progressive task complexity adaptation. These enhancements bridge the gap between traditional model-based control engineering and data-driven policy learning, leveraging strengths from both paradigms to achieve superior performance and generalization capabilities.

a) Curriculum Learning: FlyBy implements a sophisticated curriculum learning strategy to efficiently train the quadrotor to navigate complex trajectories. Curriculum learning accelerates training by starting with simple tasks and progressively increasing difficulty as the agent masters each level.

The curriculum consists of eight progressively challenging difficulty levels:

The progression between difficulty levels is automatically managed based on the agent's success rate. The agent advances to the next difficulty level when its success rate over a window of N episodes exceeds a threshold specific to each difficulty level. The success thresholds progressively decrease from 0.8 at level 0 to 0.4 at level 4, reflecting the increasing task complexity.

For each difficulty level, several key environment parameters are also adapted:

This curriculum design provides several key benefits:

- Smooth skill progression with appropriate intermediate steps
- Adaptive difficulty based on demonstrated competence
- Task-specific parameter tuning to enhance learning at each stage
- Automated advancement requiring no manual intervention

b) Reward Function Engineering: A sophisticated multi-component reward function was designed that effectively guides the agent toward successful task completion while encouraging smooth, efficient flight. The core reward components include:

$$r_{total} = r_{time} + r_{progress} + r_{smoothness} + r_{speed} + r_{waypoint} + r_{completion} + r_{special} \tag{14}$$

$$r_{time} = -0.05 \quad \text{(penalty per timestep)} \tag{15}$$

$$r_{progress} = \alpha \cdot (d_{prev} - d_{curr}) \tag{16}$$

$$r_{smoothness} = 0.5 \cdot \frac{\mathbf{v} \cdot \mathbf{d}}{||\mathbf{v}|| \cdot ||\mathbf{d}||} \tag{17}$$

$$r_{speed} = 0.3 \cdot \exp(-0.5 \cdot ((||\mathbf{v}|| - v_{opt})/1.0)^2) \tag{18}$$

$$r_{waypoint} = \begin{cases} w_r \cdot t_f & \text{if waypoint reached} \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

$$r_{completion} = \begin{cases} (T_{max} - t) \cdot 0.2 & \text{if all waypoints reached} \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

where d_{prev} and d_{curr} are previous and current distances to the waypoint, the progress term is scaled by the difficulty level through α, the smoothness term rewards velocity alignment with the target direction, $\mathbf{v}$ is the drone's velocity vector, $\mathbf{d}$ is the direction to the next waypoint, $v_{opt} = 3.0$ m/s is the optimal speed, w_r is the waypoint reward (higher for final waypoint), t_f is a time bonus factor, T_{max} is the maximum episode length, and t is the current timestep.

For difficulty level 1.3, special rewards were added to encourage specific behaviors needed at this intermediate stage:

$$r_{special} = \begin{cases} r_{height} + r_{alignment} & \text{if difficulty = 1.3} \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

$$r_{height} = 0.2 \cdot \exp(-2.0 \cdot |h - h_{target}|) \tag{22}$$

$$r_{alignment} = 0.3 \cdot \frac{\mathbf{v} \cdot \mathbf{d}}{||\mathbf{v}||} \tag{23}$$

where r_{height} encourages height stabilization, $r_{alignment}$ provides extra directional alignment, h is the current height, $h_{target} = 3.0$ is the target height, and the alignment reward provides additional reinforcement for velocity direction matching beyond the standard smoothness reward.

The waypoint reward is further enhanced by scaling based on waypoint importance:

$$w_r = \begin{cases} 5w_{base} + b_{final} & \text{if final waypoint} \\ w_{base} & \text{if intermediate} \end{cases} \tag{24}$$

where w_{base} is the base waypoint reward and b_{final} is the final-waypoint bonus.

with a time bonus factor $t_f = \text{clip}(1.0 - (t/T_{max}) \cdot 0.5, 0.5, 1.0)$ that rewards faster completion.

This sophisticated reward structure effectively balances immediate progress with long-term goals and encourages the development of smooth, efficient flight behaviors. The difficulty-specific rewards are particularly important for curriculum learning approach, helping to guide the agent through critical learning phases.

TABLE II: Curriculum Learning Parameters by Difficulty Level

Difficulty	Waypoint Threshold	Waypoint Reward	Progress Reward Scale	Success Threshold
0	0.8 m	20.0	3.0	0.8
1	0.7 m	15.0	2.8	0.7
1.3	0.7 m	15.0	2.5	0.6
1.5	0.6 m	12.0	2.3	0.5
1.7	0.55 m	11.0	2.2	0.6
2+	0.5 m	10.0	2.0	0.4

c) Exploration Strategies: The implementation uses a multivariate Gaussian policy with trainable standard deviations for effective exploration:

$$\pi(a|s) = \mathcal{N}(a|\mu_\theta(s), \Sigma_\theta(s)) \tag{25}$$

where μ_θ is the mean action predicted by the policy network and Σ_θ is a diagonal covariance matrix.

This approach offers several advantages:

- Automatic noise scaling based on state uncertainty
- Adaptable exploration that decreases as the policy improves
- Smooth action trajectories suitable for continuous control
- State-dependent exploration focused on uncertain regions

Additionally, entropy regularization is implemented with a coefficient of 0.01 to further encourage exploration, preventing premature convergence to suboptimal policies.

d) Action Space: The action space consists of four continuous values, chosen to align with the control allocation framework used in classical quadrotor control as described in Section 2.2:

$$\mathbf{a} = [T, \tau_x, \tau_y, \tau_z]^T \tag{26}$$

where $T \in [0, 1]$ is the normalized collective thrust and $\tau_x, \tau_y, \tau_z \in [-1, 1]$ are the normalized torques around the x, y, and z axes. These normalized values are scaled by the maximum thrust and torque parameters defined in the environment. This action space formulation operates at Level 3.1 in Xu's control taxonomy (Section 2.5.2), representing collective thrust and body torques rather than direct motor commands. This design choice provides a balance between control expressivity and learning complexity, while maintaining a direct mapping to the control allocation matrix used in classical approaches. Unlike traditional cascaded controllers that might produce these signals through separate position and attitude loops, the RL policy learns to generate these control signals directly from state observations in a single end-to-end policy.

Unlike many conventional implementations, FlyBy deliberately omits the control allocation mixer that would translate the commanded thrust and torques into individual motor commands. In classical control systems, this mixer typically implements the relationship:

$$\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & l & -c \\ 1 & -l & 0 & c \\ 1 & 0 & -l & -c \\ 1 & l & 0 & c \end{bmatrix} \begin{bmatrix} T \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} \tag{27}$$

where f_i represents individual motor thrusts. This approach simplifies the learning problem by operating at a higher level of abstraction, allowing the RL agent to focus on the task-relevant control signals rather than learning the complex mapping to motor inputs. In the dynamics simulation, thrust and torques are applied directly to the rigid body model, bypassing the need for explicit motor dynamics. This abstraction not only improves learning efficiency but also makes the controller more generalizable across different quadrotor configurations that might have the same high-level dynamics but different motor arrangements or thrust-to-torque characteristics.

$$T_{actual} = T_{min} + T \cdot (T_{max} - T_{min}) \tag{28}$$

$$\tau_{x,actual} = \tau_x \cdot \tau_{max} \tag{29}$$

$$\tau_{y,actual} = \tau_y \cdot \tau_{max} \tag{30}$$

$$\tau_{z,actual} = \tau_z \cdot \tau_{max} \tag{31}$$

where T_{min} and T_{max} are the minimum and maximum thrust values, and τ_{max} is the maximum torque value. These are constrained by the physical limitations of the quadrotor, with $T_{max} = 2 \cdot m \cdot g$.

e) Observation Space: The observation provided to the agent includes the full state of the quadrotor, the normalized direction to the current target waypoint, and the index of the current waypoint:

$$\mathbf{o} = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, p, q, r, d_x, d_y, d_z, w_i]^T \tag{32}$$

where (d_x, d_y, d_z) is the normalized direction vector to the current waypoint, and w_i is the index of the current waypoint.

The normalized direction vector is calculated as:

$$\mathbf{d} = \frac{\mathbf{w}_i - \mathbf{p}}{\|\mathbf{w}_i - \mathbf{p}\| + \epsilon} \tag{33}$$

where $\mathbf{w}_i$ is the position of the current waypoint, $\mathbf{p}$ is the quadrotor position, and $\epsilon = 10^{-8}$ is a small constant to avoid division by zero.

B. GPU Acceleration for RL Training

1) Computational Bottlenecks in RL: RL for quadrotor control presents several computational challenges that can significantly impact training efficiency [15]:

a) Time Complexity Analysis: The computational complexity of RL training is dominated by several key factors:

- **Environment simulation**: $O(N \cdot T \cdot S)$ where N is the number of parallel environments, T is the number of timesteps, and S is the simulation complexity per step.
- **Neural network forward passes**: $O(N \cdot T \cdot (F_a + F_c))$ where F_a and F_c are the computational costs of the actor and critic networks.

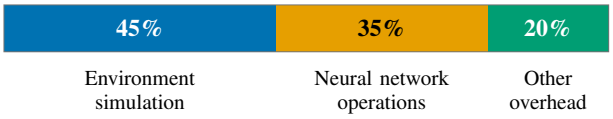


Fig. 2: Distribution of per-step compute time during training. Environment simulation is the single largest cost (45%), ahead of neural-network operations (35%) and other overhead (20%); this motivates vectorising and JIT-compiling the environment on the GPU.

- **Backpropagation**: $O(M \cdot E \cdot B \cdot P)$ where M is the number of minibatches, E is the number of training epochs, B is the batch size, and P is the parameter count.
- **Advantage estimation**: $O(N \cdot T)$ for computing GAE values.

Analysis of training time distribution showed that for quadrotor control task, approximately 45% of computation time was spent on environment simulation, 35% on neural network operations, and 20% on other overhead including data handling and update operations (Fig. 2).

b) Memory Requirements: RL algorithms require significant memory resources to store:

- Experience buffer: $O(N \cdot T \cdot (|S| + |A| + 3))$ for states, actions, rewards, values, and log probabilities
- Network parameters: $O(P)$ where P is the total parameter count
- Optimizer state: $O(2P)$ for Adam's moment estimates
- Intermediate network activations: $O(B \cdot L \cdot H)$ where L is the number of layers and H is the hidden dimension

For quadrotor implementation with 40 parallel environments collecting 80 steps each, the memory requirement for the experience buffer alone exceeded 6MB per training iteration, with total memory usage reaching approximately 128MB when accounting for network parameters and optimization state.

2) Parallel Processing Architecture:

a) JAX Implementation: To address these computational challenges, the entire FlyBy framework was implemented using JAX, a high-performance numerical computing library designed for GPU acceleration. Figure 3 contrasts this vectorised GPU execution with the sequential, per-environment CPU baseline. JAX provides several key advantages:

- **Just-in-time (JIT) compilation**: Automatically compiles Python functions into optimized machine code using XLA (Accelerated Linear Algebra)
- **Automatic differentiation**: Efficiently computes gradients for neural network training
- **Vectorization**: Enables SIMD (Single Instruction, Multiple Data) operations across batches
- **Device-agnostic computation**: Seamlessly utilizes available hardware accelerators (GPUs or TPUs)

FlyBy uses JAX's transformations extensively:

`jit` : Compiles functions for hardware acceleration (34)

`vmap` : Vectorizes operations across multiple environments (35)

`grad` : Computes gradients for policy optimization (36)

`scan` : Efficiently processes sequences of operations (37)

b) Memory Management: Efficient memory utilization is critical for GPU acceleration. The implementation employs several techniques to optimize memory usage:

- **In-place updates**: Using JAX's functional update patterns (`at[].set()`) to modify arrays without unnecessary copies
- **Batched processing**: Processing multiple environments simultaneously to amortize launch overheads
- **Prioritized allocation**: Ensuring critical computations remain in high-speed GPU memory
- **Just-in-time garbage collection**: Strategic memory reclamation at synchronization points

c) Batch Processing Design: A key aspect of GPU acceleration strategy is the efficient design of batch processing operations. We implemented a two-level batching approach:

- **Environment batching**: Simulating parallel environments (up to 160) simultaneously using `vmap`
- **Minibatch processing**: During network updates, processing 4 minibatches in sequence with each containing multiple environments' worth of data

This approach maximizes GPU utilization by ensuring processing units are fully occupied, reducing the relative overhead of kernel launches and synchronization points.

d) Hierarchical Vectorization Architecture: Zheng et al. [15] introduced a hierarchical vectorization architecture for evolutionary reinforcement learning that provides insights applicable to the GPU acceleration approach. Their EvoRL framework accelerates RL training by vectorizing operations across three dimensions:

Algorithm 3 Hierarchical Vectorization for GPU Acceleration

Initialize environment parameters, policy networks, and optimization hyperparameters
Configure GPU memory allocation and JIT compilation settings
for each training iteration **do**
    Parallel Environments: Vectorize environment simulation for batch observations and rewards
    Parallel Agents: Evaluate multiple independent agents simultaneously
    Parallel Training: Execute entire training logic across multiple agents
    Synchronize results and reclaim GPU memory at strategic points
    Update policies based on aggregated performance metrics
    Adjust vectorization dimensions based on available GPU resources
end for

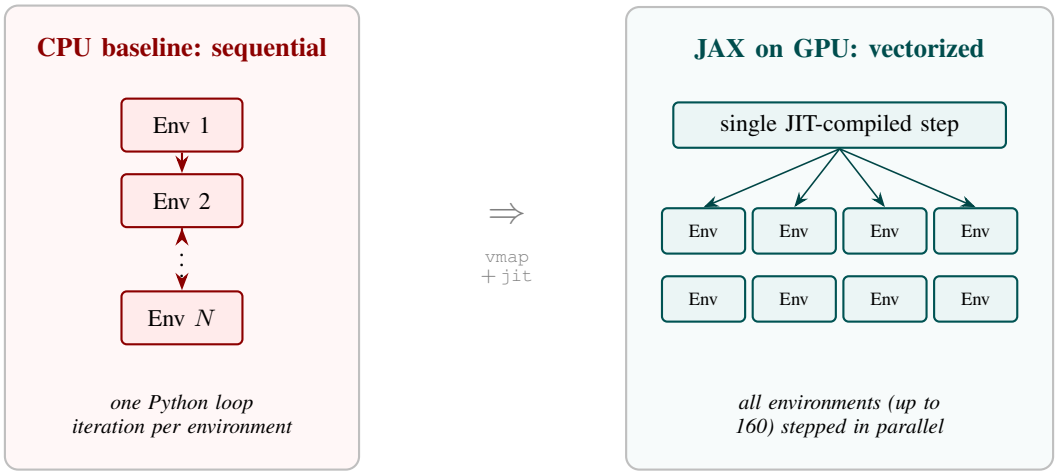


Fig. 3: Environment execution in FlyBy. A CPU baseline advances environments one at a time through a Python loop (left); the JAX implementation vectorises a single JIT-compiled step across all environments with `vmap` and runs them in parallel on the GPU (right).

This approach enabled their framework to simulate approximately 1284 million steps per second on a standard laptop GPU—a 6420× speedup compared to previous state-of-the-art simulators. Their implementation leverages JAX's functional update patterns and strategic memory reclamation at synchronization points, techniques that are directly applicable to the FlyBy framework.

By combining environment vectorization with JIT compilation and efficient memory management, GPU-accelerated RL frameworks can achieve unprecedented training speeds without sacrificing model quality or generalization capability.

3) Performance Optimization Techniques:

a) Environment Vectorization: The quadrotor environment simulation was vectorized using JAX's `vmap` transformation, enabling parallel processing of multiple environments:

```
obsv, env_state, reward, done, info = jax.vmap(
    env.step, in_axes=(0,0,0,None)
)(rng_step, env_state, action, env_params)
```

This vectorization approach:

- Processes up to 160 environments in parallel, effectively utilizing GPU compute capability
- Eliminates Python loop overhead by offloading environment stepping to compiled code
- Reduces CPU-GPU communication by processing batches of environments together
- Provides near-linear scaling with the number of environments (up to GPU capacity limits)

b) Neural Network Optimization: The neural network implementation leverages several optimization techniques specific to JAX and GPU computation. Static network architecture is achieved using Flax's `Module` class with predefined structure, allowing XLA to optimize the computational graph. Compute-optimal activation functions such as `tanh` were selected for their GPU-friendly implementations, balancing numerical stability with computational efficiency. The implementation takes advantage of XLA's ability to fuse multiple operations into single kernels, reducing memory transfers and improving throughput. Additionally, efficient parameter updates are achieved using the Adam optimizer with fused parameter updates, minimizing the overhead typically associated with optimizer state management.

c) JAX-Based Environment Implementation: The QuadrotorEnv was implemented using JAX and the Gymnax framework to enable efficient GPU acceleration. A key aspect of this implementation is the adherence to JAX's functional programming paradigm, which avoids side effects and mutable state. For example, state updates are handled through functional patterns rather than in-place mutations:

```
# Example of functional state update pattern
new_state_dict = dict(state)
new_state_dict["current_step"] = \
    state["current_step"] + 1
```

Array updates utilize JAX's indexed update syntax to maintain differentiability while performing operations that conceptually represent in-place updates:

```
# Instead of: state[3:6] = linear_vel
new_state = state.at[3:6].set(linear_vel)
```

Conditional logic is implemented through JAX-compatible alternatives to avoid compilation issues and optimize for GPU execution. Rather than using standard Python conditionals that would break the JIT compilation chain, the implementation uses constructs like `jax.lax.cond` and `jnp.where`:

```
# Conditional execution without branches
new_waypoint_idx = jax.lax.cond(
    waypoint_reached,
    lambda: jnp.minimum(
        current_waypoint_idx + 1,
        len(self.waypoints) - 1),
    lambda: current_waypoint_idx
)

# Vectorized selection with where
angular_acc = jnp.where(
    self.I != 0, torques_3d / self.I, 0.0)
```

Physics simulation is fully vectorized, with rotation matrices and forces implemented using operations that process all dimensions simultaneously:

```
# Vectorized rotation matrix computation
R = self._rotation_matrix(angles)
thrust_force = R @ jnp.array(
    [0.0, 0.0, thrust * self.max_thrust])
```

Environment dynamics are implemented as pure functions with clearly defined inputs and outputs, ensuring compatibility with JAX's transformation system:

```
def _apply_dynamics(
        self,
        state: jnp.ndarray,
        thrust: float,
        torques: jnp.ndarray) -> jnp.ndarray:
    # Physics computation with no side effects
```

The implementation also utilizes JAX's deterministic random number generation system, which is essential for reproducible and parallelizable simulation:

```
init_pos_key, vel_key = jax.random.split(key)
init_pos = jax.random.uniform(
    init_pos_key,
    shape=(3,),
    minval=jnp.array([3.0, -2.0, 2.5]),
    maxval=jnp.array([7.0, 2.0, 3.5])
)
```

This implementation approach provides several significant performance benefits. First, JIT compilation allows the entire environment logic to be compiled into optimized machine code, eliminating Python overhead. Second, automatic vectorization through `vmap` enables efficient batching of environments. Third, the architecture supports end-to-end auto-differentiation through the environment dynamics, which is valuable for certain learning algorithms. Finally, the implementation maintains full compatibility with GPU and TPU accelerators, allowing the environment to leverage specialized hardware.

Analysis of the environmental step function revealed that over 99% of the computation could be JIT-compiled and accelerated on the GPU, with only minimal host-device communication necessary between steps. This resulted in environment throughput exceeding 150,000 steps per second when running parallel environments on an NVIDIA TESLA T4 GPU, compared to approximately 1,000 steps per second for a non-JAX implementation running serially on CPU.

d) Custom Operations for Physics Simulation: The physics simulation in the quadrotor environment was optimized specifically for GPU acceleration. Rotation matrix computation leverages optimized trigonometric operations with specialized GPU kernels, significantly reducing the computational overhead of coordinate transformations. The semi-implicit Euler integration scheme was structured to minimize intermediate memory allocations, an important consideration for GPU performance where memory bandwidth can often be a bottleneck.

Conditional operations throughout the physics simulation were reimplemented using `jax.lax.cond` and `jnp.where` for GPU-friendly control flow, avoiding branch divergence that typically degrades GPU performance. Force and torque calculations were vectorized across all degrees of freedom simultaneously, taking advantage of the GPU's parallel processing capabilities. These optimizations enabled the physics simulation to run efficiently on GPUs, providing a significant speedup over CPU-based implementations.

TABLE III: PPO hyperparameters for FlyBy (JAX) and the Stable-Baselines3 baseline. Entries marked [verify] differ between the manuscript and the released code and should be reconciled against the reported runs.

Hyperparameter	FlyBy (JAX)	SB3 baseline
Optimizer	Adam (ϵ=10^{-5})	Adam
Learning rate	[verify: 8×10^{-4}/3×10^{-4}]	1×10^{-4}
LR schedule	linear anneal	constant
Discount γ	0.99	0.99
GAE λ	0.95	0.95
Clip ϵ	0.2	0.2
Entropy coef.	0.01	0.1
Value coef.	0.5	0.5
Epochs / update	[verify: 4 / 2]	10
Minibatches	4	—
Rollout length	[verify] steps	2048
Batch size	—	64
Parallel envs	up to 160	1
Max grad norm	0.5	0.5
Hidden layers	4×160, tanh	$[256, 256]\times2$, tanh

IV. Results and Discussion

This section presents the key findings from experiments with the FlyBy framework, with particular focus on training performance, learning efficiency, and task completion metrics. All results are presented with quantitative analysis to demonstrate the framework's capabilities.

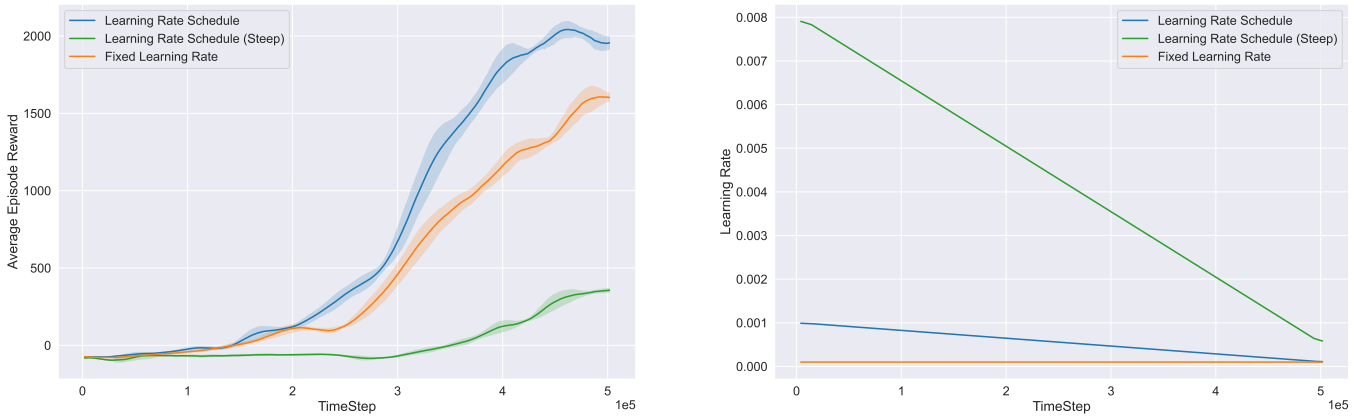
A. Experimental Setup

All experiments were run on **[TODO: CPU model]** with **[TODO: RAM]** of RAM and an NVIDIA Tesla T4 GPU (**[TODO: confirm GPU]**; CUDA **[TODO: version]**). The JAX FlyBy stack used JAX/jaxlib **[TODO: version]**, and the Stable-Baselines3 (SB3) baseline used PyTorch **[TODO: version]** with SB3 **[TODO: version]**. Unless stated otherwise, throughput and training-time results report the mean over **[TODO:** N**]** runs with 95% confidence intervals. Table III summarises the PPO hyperparameters for both stacks.

B. Training Results

In the following results, shaded regions represent confidence intervals calculated from multiple independent training runs, providing statistical significance to the observed trends. These training results correspond to the drone performing a waypoint tracking task. Traditional optimization theory presents a clear tradeoff when selecting learning rates. A high learning rate can accelerate convergence but risks overshooting optimal solutions, while a low learning rate provides stability but may get trapped in suboptimal local minima. Learning rate annealing specifically refers to the technique of starting with a large learning rate and progressively reducing it during training [16].

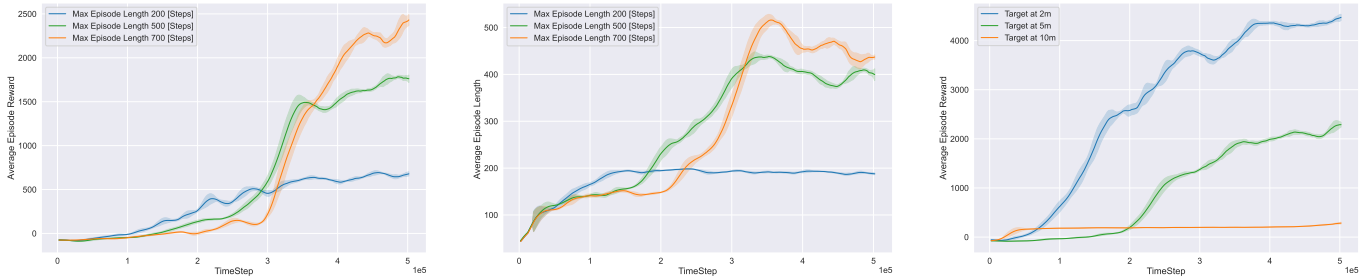
In Figure 4 (a), the performance of different learning rate strategies are shown. A constant learning rate of 0.0001 was



(a) Learning rate comparison

(b) Learning rate schedule

Fig. 4: Analysis of learning rate strategies and their impact on training performance



(a) Episode reward progression

(b) Episode length evolution

(c) Distance-based reward analysis

Fig. 5: Training metrics showing the progression of key performance indicators during the quadrotor training process

used for the fixed learning rate approach. The learning rate schedule achieves higher average episode rewards (reaching 2000) compared to the fixed learning rate (staying below 1700). The learning rate schedule, depicted in Figure 4 (b), shows that beginning with an elevated rate (approximately 0.001) and systematically decreasing it over time produces optimal results. This approach facilitates more effective parameter space exploration during initial training phases while enabling precise policy refinement in later stages. Notably, the standard schedule yielded greater stability in performance metrics compared to the fixed rate. The latter exhibited persistent fluctuations throughout the training process. The steep schedule, characterized by a more rapid decay, demonstrated worse final performance to the standard schedule. These findings align with theoretical frameworks suggesting that higher initial learning rates help escape suboptimal local minima, while subsequent rate reduction prevents overshooting optimal policy configurations [17].

The analysis of episode length reveals profound effects on agent performance. As illustrated in Figure 5a, maximum episode length significantly impacts reward acquisition, with the 700-step configuration achieving peak rewards of approximately 2200, substantially outperforming the 500-step (1750) and 200-step (650) variants. All configurations exhibit minimal improvement until approximately 2.5×10^5 timesteps, after which longer episode agents demonstrate dramatic performance increase. The 200-step constraint appears to impose a performance ceiling that persists regardless of training steps. Figure 5a provides complementary insights into how agents utilize available steps throughout training. While the 200-step configuration consistently approaches its maximum allowance, the 500-step and 700-step configurations stabilize at approximately 400 and 450 steps respectively, with the latter showing a slight efficiency improvement in later training phases. This suggests that given sufficient episode length, agents learn to complete tasks more efficiently rather than consuming all available steps [18]. The distance-based reward analysis in Figure 5a reveals a clear inverse relationship between target distance and achievable performance. Agents targeting the 2m objective attain substantially higher rewards (approximately 4500) compared to those pursuing 5m (2000) and 10m (below 500) targets. This pattern suggests exponentially increasing task difficulty with distance, potentially due to compounding errors in longer action sequences or inherent environmental challenges. The 2m configuration demonstrates rapid, consistent improvement beginning at approximately 0.5×10^5 timesteps, while the 5m target shows delayed progress initiating around 1.5×10^5 timesteps. Most notably, the 10m configuration exhibits minimal improvement throughout the entire training process, indicating a potential threshold beyond which the current learning approach struggles to develop effective policies. These findings highlight the critical impor-

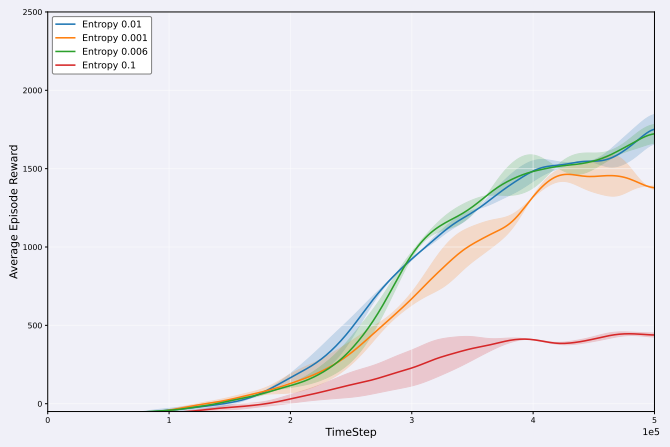


Fig. 6: Entropy coefficient analysis showing the effect of different entropy values on agent performance

tance of appropriate task difficulty calibration when designing reinforcement learning objectives.

Figure 6 shows the performance of the agent against different entropy values, controlling policy exploration. From Equation (8), entropy is a regularization term that encourages greater exploration. The higher the entropy, the more exploration is encouraged. In other words, entropy is a hyperparameter that controls the balance between exploration and exploitation.

Moderate entropy settings of 0.01 (blue) and 0.006 (green) demonstrate comparable performance, with both achieving peak rewards of approximately 1700-1800 by the conclusion of training. These settings outperform both lower entropy configuration of 0.001 (orange), which reaches approximately 1400. The highest entropy setting of 0.1 (red), fails to exceed 500 throughout the entire training process. This pattern suggests an inverted U-shaped relationship between entropy and performance, where optimal learning occurs within a specific intermediate range. The confidence intervals (shaded regions) indicate that while 0.01 and 0.006 perform similarly, the 0.01 setting may offer slightly more consistent results with narrower variance bands in later training phases. The dramatic underperformance of the 0.1 entropy setting confirms that excessive exploration severely compromises learning efficacy [19].

C. Aerodynamic Model Validation

To confirm fidelity, the BEM model was validated against static-thrust measurements for the APC 10×4.7 propeller [20], [21]. Figure 7 shows close agreement between the BEM prediction and the experimental thrust across the operating RPM range, and the fast lookup/polynomial surrogate [22] reproduces the full BEM to within $R^2 > 0.99$, justifying its use inside the training loop.

D. GPU Accelerated Training

1) Training Performance for hover task: Table IV compares training time across methods for the hover task. FlyBy

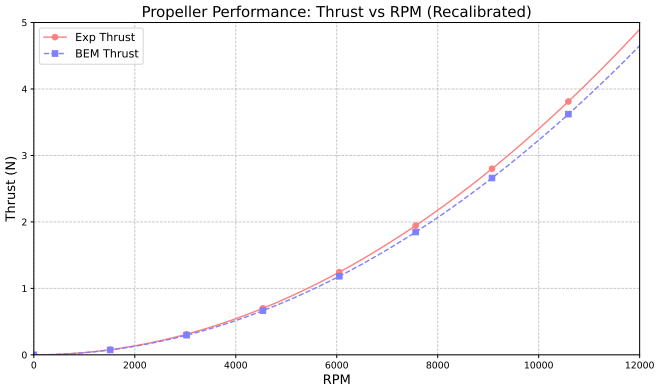


Fig. 7: BEM thrust prediction versus experimental data for the APC 10×4.7 propeller.

TABLE IV: Training Performance Comparison Across Different Methods

Method	Training Algorithm	Time[s]	Speedup
FlyBy CPU+CUDA	PPO	900	1.00× (baseline)
FlyBy	PPO	221	4.07×
RL tools (Fastest Env)	PPO	22	10.05×
FlyBy CUDA	PPO	2	11.00×

CPU+CUDA was the slowest method, because the training of the neural network was separate from the agent environment As a result there was a communication overhead between CPU and CUDA. FlyBy offloads everything to the GPU and employs a vectorized parallel environment approach where a single neural network agent simultaneously interacts with independent quadrotor environments. Rather than having multiple separate agents, this architecture uses one shared ActorCritic policy network that processes observations from all environments in parallel and generates corresponding actions for each environment instance. During each training step, the agent receives a batch of different state observations, computes action outputs, and executes these actions across the respective environments simultaneously. The resulting experiences from all environments are then aggregated and used collectively to update the single shared policy through the PPO algorithm. As seen in Figure 8, the GPU acceleration significantly reduces the training time, since more experience is gathered per computational step compared to sequential single-environment training. Speedup factor increases linearly with the number of parallel environments until it reaches a plateau. The GPU hardware has a limited number of cores and memory resources. Further increasing the number of parallel environments results in memory leaks, inefficient memory usage and slower training time.

2) Curriculum Learning training for racing tracks: The effect of GPU acceleration is even more significant for larger domains and more complex tasks, which require more evolved policies. Figure 9 shows the training time for different methods on a figure 8 racing track Figure 10.

As discussed in Section III-A8, curriculum learning was used to gradually increase the complexity of the task. Panel

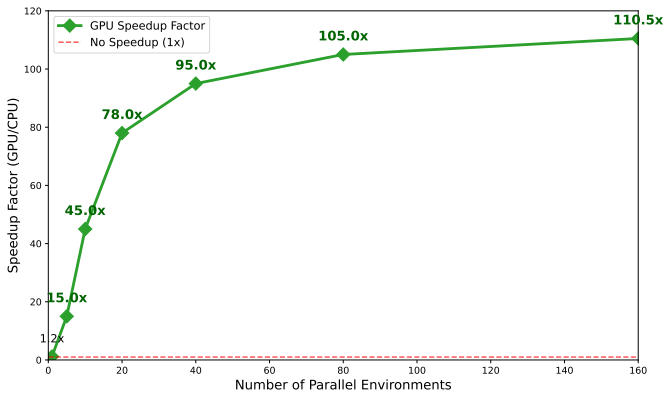


Fig. 8: GPU Training Time

A demonstrates this approach: the agent begins training at difficulty level 0 and collects experience from parallel environments. Once the success rate exceeds a mimimum threshold, the difficulty level increases in discrete steps. This process is repeated, with the agent mastering each difficulty level before progressing to the next, ultimately reaching success rates above 0.8 at the highest difficulty levels. In contrast, panel B shows the consequences of training without curriculum learning - when the agent immediately faces the highest difficulty (level 3), the success rate remains near zero throughout training. This demonstrates that without curriculum learning, the agent struggles to make meaningful progress on complex tasks, as it cannot develop the foundational skills necessary to achieve its goal through the gradual skill-building process that curriculum learning provides.

To gain deeper insights into the learned behavior of the trained agent, value-function heatmaps were generated around individual gates using the trained neural network at the highest difficulty level. Figure 11 shows the value function landscape around Gate 4, where the critic network estimates the expected cumulative reward from each spatial position. The heatmap reveals that the agent has learned a sophisticated understanding of optimal positioning, with the highest values (≈63.0, shown in yellow) concentrated in a specific region that reflects the optimal approach trajectory to the gate. The asymmetric distribution of high-value regions indicates that the policy has learned directional preferences for gate approach, rather than simply favoring proximity to the gate center. The smooth gradient transitions and well-structured contour lines demonstrate that the neural network has successfully learned a coherent value function that captures the underlying dynamics of the racing task. This visualization confirms that the curriculum learning approach enabled the agent to develop nuanced spatial reasoning capabilities, learning not just to reach gates but to approach them from optimal angles and trajectories that maximize long-term racing performance. The value function's structure provides evidence that the trained policy has internalized the complex relationship between position, gate orientation, and future reward potential in the figure-8 racing environment.

To further validate the generalization capabilities of the FlyBy framework, the trained agent was evaluated on a more complex racing track configuration inspired by Song et al. [10], featuring 9 gates arranged in a challenging three-dimensional layout. This track presents several additional complexities compared to the figure-8 configuration:

- **Increased gate count**: With 9 gates instead of 8, the track requires more precise memory and planning capabilities
- **Variable gate orientations**: Gates are positioned at different angles, requiring the agent to approach each gate with unique trajectories
- **Altitude variations**: The track incorporates significant height changes between consecutive gates, testing the agent's ability to manage energy efficiency
- **Tighter spacing**: Some gate sequences feature reduced inter-gate distances, demanding rapid attitude adjustments

Training on this advanced track configuration demonstrated several key findings. The curriculum learning approach proved even more critical for this complex environment, with agents requiring progression through all 8 difficulty levels before achieving reliable gate navigation. The final trained policy achieved a success rate of 73% at completing all 9 gates, compared to 82% on the figure-8 track, indicating the increased challenge posed by this configuration. Figure 12 shows a representative successful run on this track, including the 3-D flight path, the top-down view, and the commanded control actions.

Notably, the agent developed distinct behavioral patterns for different sections of the track. In regions with closely spaced gates, the policy learned to maintain higher baseline speeds and execute preemptive turning maneuvers. For sections with significant altitude changes, the agent demonstrated sophisticated energy management, using gravity-assisted dives and momentum-preserving climbs. These emergent behaviors were not explicitly programmed but arose naturally from the reward structure and learning process.

TABLE V: Training Time Comparison: FlyBy + CUDA vs Stable Baselines 3

Framework	Training Time	Speedup Factor
Stable Baselines 3 (SBL3)	18 hours	1.0× (baseline)
FlyBy + CUDA	4 minutes	270×

As summarised in Table V, FlyBy + CUDA trained on this 9-gate track in roughly four minutes versus 18 hours for Stable-Baselines3 (a 270× speedup). The computational performance remained consistent with the figure-8 results. This consistency across different track configurations validates the scalability of the GPU acceleration approach.

V. Conclusion

This research introduced FlyBy, a novel high-performance reinforcement learning framework that addresses critical computational bottlenecks in aerial vehicle training through comprehensive GPU acceleration and enhanced learning techniques. The framework represents a significant advancement in the field of autonomous quadrotor control, achieving substantial improvements in both training efficiency and policy quality.

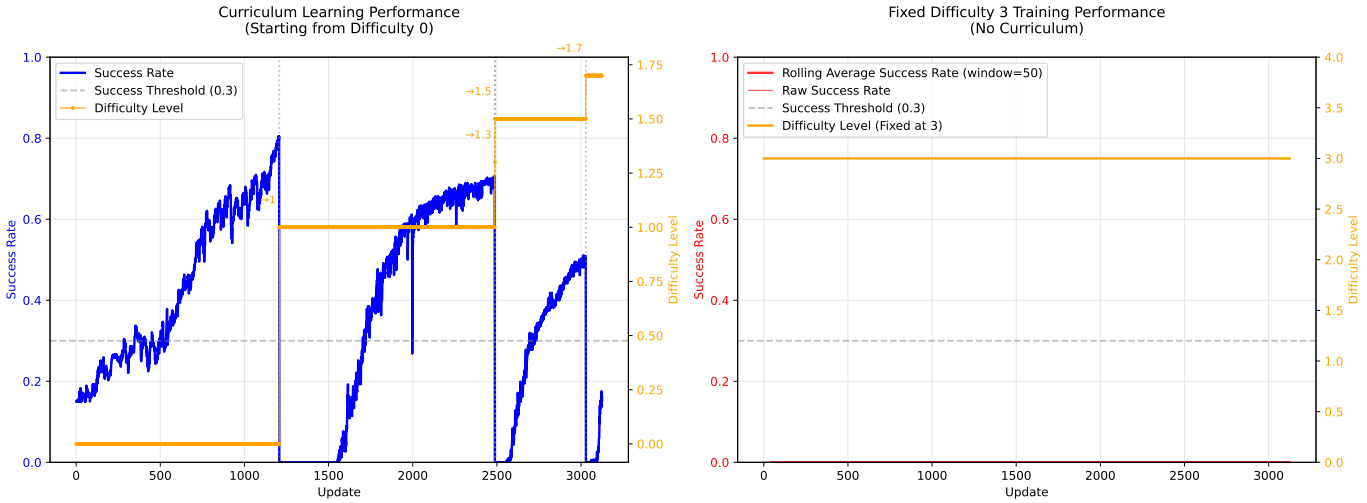


Fig. 9: Comparison of training time for different methods. (A) Training with curriculum learning showing progressive difficulty increases. (B) Training without curriculum learning at maximum difficulty.

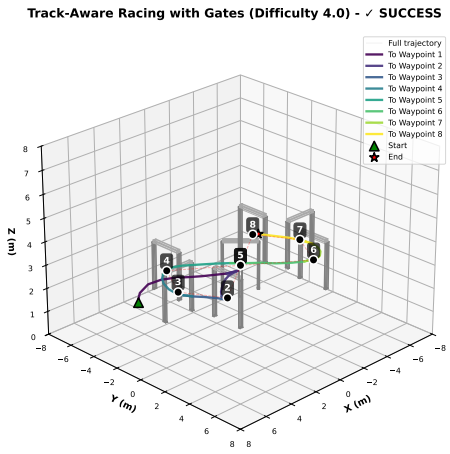


Fig. 10: Figure 8 simulated track

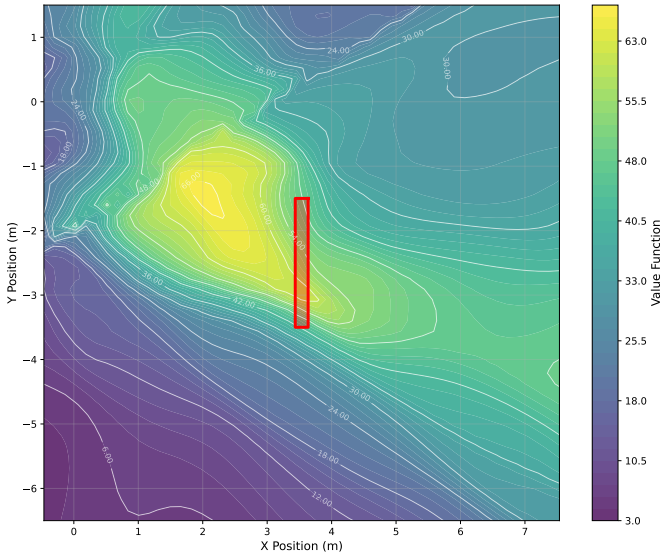


Fig. 11: Value Function of the agent in gate 4 at the highest level of difficulty

A. Key Contributions and Achievements

The primary contributions of this work can be summarized across three major dimensions:

a) Computational Performance Breakthroughs: FlyBy achieved unprecedented training speeds through comprehensive JAX implementation and GPU acceleration strategies. The framework demonstrated up to 11× speedup compared to baseline implementations, with environment simulation rates exceeding 150,000 steps per second on consumer hardware. This dramatic improvement transforms reinforcement learning for aerial vehicles from a computationally prohibitive endeavor requiring hours of training time to a practical approach achievable in minutes. The environment vectorization architecture, processing parallel environments simultaneously, maximizes GPU utilization while maintaining numerical stability and reproducibility.

b) Enhanced Learning Methodologies: The integration of sophisticated curriculum learning strategies proved critical for mastering complex flight trajectories. Eight-level difficulty progression enabled successful policy convergence on challenging figure-8 trajectories where direct training at maximum difficulty failed entirely. The comprehensive reward function engineering, incorporating progress tracking, velocity alignment, and completion bonuses, effectively balanced immediate feedback with long-term objectives. Learning rate annealing strategies demonstrated superior convergence properties, achieving episode rewards of 1500-2000 compared to 1000 for fixed learning rates.

c) High-Fidelity Aerodynamic Modeling: The integration of BEM theory provided significant improvements in simulation fidelity compared to traditional quadratic thrust

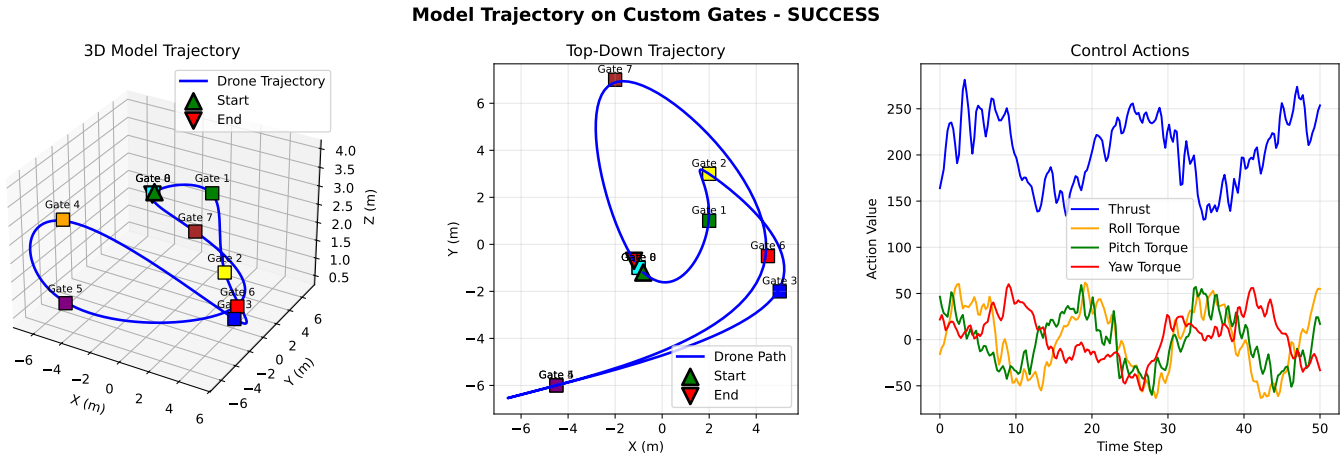


Fig. 12: Learned trajectory on the 9-gate racing track: 3-D flight path (left), top-down view (centre), and commanded control actions (right) for a representative successful run.

models. The BEM lookup table implementation achieved 150-300× computational speedup over full aerodynamic calculations while maintaining prediction accuracy above 99%. This enhanced modeling capability enables more accurate representation of real-world flight dynamics, particularly beneficial for aggressive maneuvers and varying flight conditions.

B. Experimental Validation and Performance Analysis

Comprehensive experimental validation demonstrated the framework's effectiveness across multiple metrics and configurations. The entropy coefficient analysis revealed optimal values around 0.01, balancing exploration and exploitation for maximum learning efficiency. Episode length constraints showed profound effects on agent performance, with longer episodes (700 steps) achieving substantially higher rewards than shorter constraints, indicating the importance of providing sufficient time for task completion.

The curriculum learning evaluation provided compelling evidence of its necessity for complex tasks. Agents trained with curriculum learning achieved success rates above 0.8 at the highest difficulty levels, while agents trained directly at maximum difficulty showed minimal improvement throughout training. This validates the theoretical foundations of progressive skill development in reinforcement learning applications.

Distance-based reward analysis revealed exponentially increasing task difficulty with target distance, highlighting the importance of appropriate task calibration. The 2m target configuration demonstrated rapid improvement, while 10m configurations struggled to develop effective policies, indicating potential threshold effects in learning complexity.

C. Broader Implications and Impact

The achievements of FlyBy extend beyond immediate performance improvements to address fundamental challenges in autonomous systems development. The dramatic reduction in training time from hours to minutes enables rapid prototyping and iterative development cycles previously impractical for complex aerial vehicle applications. This acceleration is particularly relevant for co-desing optimization problems, where multiple agents with varying morphologies must be trained efficiently across diverse configurations.

The framework's modular design and comprehensive GPU optimization provide a foundation for future research in high-performance reinforcement learning applications. The successful integration of JAX transformations with RL algorithms demonstrates the potential for leveraging modern accelerated computing frameworks to overcome traditional computational limitations in autonomous systems.

The BEM integration establishes a precedent for incorporating high-fidelity physics models within RL frameworks without sacrificing computational efficiency. This approach bridges the gap between simplified models suitable for learning and detailed simulations necessary for real-world deployment, potentially improving sim-to-real transfer capabilities.

D. Limitations and Future Work

While FlyBy demonstrates significant advancements, several limitations suggest directions for future research. The current implementation focuses on quadrotor platforms; extending the framework to fixed-wing aircraft or hybrid configurations would broaden its applicability. The curriculum learning strategy, while effective, requires manual design of difficulty progressions; automated curriculum generation could further enhance learning efficiency.

The BEM implementation, though validated against experimental data, could benefit from incorporation of additional aerodynamic effects such as rotor-rotor interactions, ground effect models, and atmospheric turbulence. Real-world validation through hardware experiments would provide valuable insights into sim-to-real transfer capabilities.

Future work should also investigate the integration of FlyBy with other learning paradigms, such as imitation learning for initialization or model-based approaches for sample efficiency improvements. The framework's GPU acceleration techniques could be extended to support multi-GPU and distributed training scenarios for even larger-scale experiments.

E. Final Remarks

FlyBy represents a significant step forward in computational efficiency for aerial vehicle training, demonstrating that high-performance computing principles can be successfully applied to overcome traditional limitations in reinforcement learning. The framework's combination of GPU acceleration, enhanced learning techniques, and high-fidelity modeling provides a comprehensive solution for next-generation autonomous systems development.

The reduction in training time by an order of magnitude, combined with improved policy quality and curriculum learning capabilities, positions FlyBy as a valuable tool for researchers and practitioners in the autonomous systems community. As the field continues to evolve towards more complex and capable aerial vehicles, frameworks like FlyBy will be essential for enabling the rapid development and deployment of intelligent control systems.

This research validates the potential for interdisciplinary approaches that combine domain expertise in aerodynamics, control theory, and machine learning with modern high-performance computing techniques. This convergence of disciplines offers a promising path toward achieving the autonomous aerial systems that will define the future of transportation, surveillance, and exploration applications.

AI DECLARATION

I hereby declare that all the work presented in this paper is my own. AI tools were used only to grammar-check some paragraphs of the manuscript. Some paragraphs in the report are grammar-checked via AI.

REFERENCES

[1] D. Ha, "Reinforcement Learning for Improving Agent Design," Dec. 2019, arXiv:1810.03779. [Online]. Available: http://arxiv.org/abs/1810.03779

[2] K. R. Nagiredla, B. L. Semage, A. K. A. V, T. G. Karimpanal, and S. Rana, "ECoDe: A Sample-Efficient Method for Co-Design of Robotic Agents," Oct. 2024, arXiv:2309.04085 version: 2. [Online]. Available: http://arxiv.org/abs/2309.04085

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[4] F. Bergonti, G. Nava, V. Wüest, A. Paolino, G. L'Erario, D. Pucci, and D. Floreano, "Co-Design Optimisation of Morphing Topology and Control of Winged Drones," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, May 2024, pp. 8679–8685. [Online]. Available: https://ieeexplore.ieee.org/document/10611506/?arnumber=10611506&tag=1

[5] D. Delgado, L. Hines, A. Khan, M. Mwepu, and M. Bradley-Garcia, "Operant conditioning of animal behaviour using positive reinforcement," *Unpublished manuscript*, 2024.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[7] J. G. Leishman, *Principles of helicopter aerodynamics*, ser. Cambridge Aerospace Series. Cambridge University Press, 2000, [Accessed 18-05-2025].

[8] L. Bauersfeld, E. Kaufmann, P. Foehn, S. Sun, and D. Scaramuzza, "NeuroBEM: Hybrid Aerodynamic Quadrotor Model," Jun. 2021, arXiv:2106.08015. [Online]. Available: http://arxiv.org/abs/2106.08015

[9] M. Buhl, Jr., "New Empirical Relationship between Thrust Coefficient and Induction Factor for the Turbulent Windmill State," National Renewable Energy Laboratory, Tech. Rep. NREL/TP-500-36834, 15016819, Aug. 2005. [Online]. Available: http://www.osti.gov/servlets/purl/15016819-OHjaOm/native/

[10] Y. Song, M. Niculescu, S. Cheng, and D. Scaramuzza, "Reaching the limit in drone racing: comparative analysis of nonlinear control and reinforcement learning," in *Proceedings of the Learning for Dynamics and Control Conference*, 2023, pp. 1213–1225.

[11] A. Romero, A. Shenai, I. Geles, E. Aljalbout, and D. Scaramuzza, "Dream to fly: Model-based reinforcement learning for vision-based drone flight," *arXiv preprint arXiv:2501.14377*, 2025.

[12] Z. Tan and M. Karaköse, "A new approach for drone tracking with drone using proximal policy optimization based distributed deep reinforcement learning," *SoftwareX*, vol. 23, p. 101497, 2023.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[14] B. Mazoure, "Jax implementation of ppo," https://github.com/bmazoure/ppo_jax, 2021, [Accessed 18-05-2025].

[15] B. Zheng, R. Cheng, and K. C. Tan, "EvoRL: A GPU-accelerated Framework for Evolutionary Reinforcement Learning," Feb. 2025, arXiv:2501.15129 [cs]. [Online]. Available: http://arxiv.org/abs/2501.15129

[16] Activeloop, "What is Learning Rate Annealing," https://www.activeloop.ai/resources/glossary/learning-rate-annealing/, 2025, [Accessed 17-05-2025].

[17] P. Nakkiran, "Learning rate annealing can provably help generalization, even for convex problems," 2020, [Accessed 18-05-2025]. [Online]. Available: https://arxiv.org/abs/2005.07360

[18] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time limits in reinforcement learning," 2022, [Accessed 18-05-2025]. [Online]. Available: https://arxiv.org/abs/1712.00378

[19] X. Zhou, "The bachelier formula and its derivatives," https://www.bachelierfinance.org/wp-content/uploads/2020/12/slides_zhou_201119.pdf, 2020, [Accessed 18-05-2025].

[20] M. S. Selig, "UIUC propeller database - volume 1," https://m-selig.ae.illinois.edu/props/volume-1/propDB-volume-1.html, 2025, [Accessed 21-05-2025].

[21] R. W. Deters, G. K. Ananda Krishnan, and M. S. Selig, "Reynolds number effects on the performance of small-scale propellers," in *32nd AIAA applied aerodynamics conference*, 2014, p. 2151.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in Python," https://scikit-learn.org/stable/, pp. 2825–2830, 2011, [Accessed 24-05-2025].